Insights for Exascale IO APIs from Building a Petascale IO API

Jay Lofstead
Sandia National Laboratories
gflofst@sandia.gov

Robert Ross
Argonne National Laboratory
rross@mcs.anl.gov

ABSTRACT

Near the dawn of the petascale era, IO libraries had reached a stability in their function and data layout with only incremental changes being incorporated. The shift in technology, particularly the scale of parallel file systems and the number of compute processes, prompted revisiting best practices for optimal IO performance.

Among other efforts like PLFS, the project that led to ADIOS, the ADaptable IO System, was motivated by both the shift in technology and the historical requirement, for optimal IO performance, to change how simulations performed IO depending on the platform. To solve both issues, the ADIOS team, along with consultation with other leading IO experts, sought to build a new IO platform based on the assumptions inherent in the petascale hardware platforms.

This paper helps inform the design of future IO platforms with a discussion of lessons learned as part of the process of designing and building ADIOS.

1. INTRODUCTION

IO libraries have a long history. Each was developed by a community to offer an abstraction of utility to a range of applications and offer interoperability for different users of the library. As these libraries mature, they develop a level of portability for written data and generalize the API calls for writing and reading data. The most successful of these have become standards used beyond their original community. Each of these offers different features and trade-offs. As an upside, third-party tools have a broad understanding and support for data stored using these formats. An unfortunate downside is the lack of detailed control to optimize how the data is organized. One painful example of the lengths taken by users to work around the file format can be seen in how the Chombo AMR framework [1] used HDF-5 at the time. Rather than just storing multidimensional arrays of structs or multiple arrays, the data was genericized into a list of values that had a known mapping. To get a value, one had to decode how many values were stored per element and the sizes, and then calculate which values to read. This approach achieved the desired performance and was seen as a preferable option to building a custom data layout and dealing with getting support by third-party tools. These experiences help shape future IO API designs.

Copyright 2013 Association for Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
SC13 November 17-21, 2013, Denver, CO, USA
ACM 978-1-4503-2378-9/13/11
http://dx.doi.org/10.1145/2503210.2503238

ADIOS is the latest of these libraries and has a similar history. It was designed and built originally based on science applications that use Oak Ridge's HPC platforms with some connection to the National Center for Computational Science group. Two particular applications, the Chimera supernova simulation [2] and the GTC fusion simulation [3], were the base cases used to test the designs on the Oak Ridge hardware. In order to see the motivation for creating ADIOS, it helps to understand what the other main options for performing HPC IO were at the time. There were three primary IO stack options for HPC applications that are still in wide use today: NetCDF, PnetCDF, and HDF-5.

NetCDF version 3 [4] and the CDF-3 file format were developed to support the climate community and have gained support from some other applications. The data layout consists of a header block containing all the metadata about the file contents immediately followed by the data with each variable stored in an inorder, contiguous chunk. The big limitation of NetCDF version 3 and older is that it is a serial IO library. The development of parallel storage arrays and larger numbers of compute processes and large aggregate data sizes prompted the need to move to a parallel IO library to achieve better performance. As a stop-gap measure, NetCDF 3.6.0 introduced a 64-bit offset, but it was still limited to a serial API.

PnetCDF [5] and the CDF-5 file format offered a shift to full support for 64-bit and parallel IO while maintaining as much as possible backwards compatibility with the CDF-3 file format. While PnetCDF is successful and still used, the NetCDF community ultimately opted to switch to using HDF-5 underneath the NetCDF API for their go-forward platform. In spite of the NetCDF community's choice to pursue a different direction, a level of interoperability exists between NetCDF version 4 using HDF-5 underneath, CDF-3, and the CDF-5 file format. The tools available for manipulating and accessing these files can work on the various formats.

The HDF-5 [6] API and file format were developed at the National Center for Supercomputing Applications and offers a more modular structure than CDF by using a block-based structure in the file. Instead of the strictly linear format, HDF-5 offers linking to other blocks in order to expand or contract the metadata entries and place the variables in some less predictable order. However, it still by default stores the data for each variable in a single, con-

tiguous, inorder chunk. An option allows storing the data in multiple chunks, but it requires additional configuration and decisions that few users make. HDF-5 has become the general standard for many HPC applications as well as other industries. NetCDF version 4, as well as other libraries such as SILO [7], uses HDF-5 to define and implement the mapping of data to files. This frees the designers to focus on building an API that suits their applications rather than worrying about the data layout. Moreover, HDF-5 files are broadly compatibile with third-party tools, including plugins such as automatic data compression.

The major IO innovation that drove these IO APIs for parallel performance was the introduction of two-phase IO [8]. Collectively sending data to a single file in parallel afforded taking advantage of all the storage devices simultaneously. To aid in performance, the first phase of two-phase IO is to rearrange data so that the chunks being written to disk are larger. This achieves fewer, larger writes reducing disk head movement resulting in a net benefit. The key idea is to trade disk head or platter movement for interconnect communication. The net result, hopefully, will be better IO performance.

Two-phase IO offers benefits even today for data sets meeting particular characteristics. For example, it has been shown to achieve as much as 50% of peak write performance [9] at 64K processes. The tradeoff of disk seek time and rotational latency for data rearrangement does not necessarily scale as well when the number of processes and amount of data that must be exchanged explode. One could do better by using tricks as Chombo did, but doing so requires careful profiling and adjustment of how the data is organized and the IO performed knowing the context of the IO API and the disk layout.

In that environment of 2005, a project started trying to improve the IO performance for the GTC fusion simulation. In 2007, the effort restarted with a different approach that proved effective. ADIOS was designed under the assumption of a massively parallel storage array and 3-D domain decompositions. It had a foot in the older world as well by considering the requirements of the 2-D Chimera code available at the time that used a large number of smaller variables. ADIOS focused initially on writing performance demonstrating saturating the IO system for a variety of workloads [10]. Later, the "write optimized" format was validated to not penalize reading. In almost all cases, it yielded better reading performance for representative reading patterns [11, 12].

Today, the technology is shifting again. NAND-based Flash storage's rapid decline in price and increase in performance is threatening the dominance of disk. In the near future, other technologies such as phase change memory will change this environment further. Not only will the nature of the storage devices change, but also the number of levels and location of each level physically will change. The energy envelope limitations for exascale platforms are also putting pressure on changing how IO is performed. The IO APIs need to consider these and other technology shifts in order to better meet the needs of their customers.

This paper presents lessons for developers of next generation IO APIs inspired by the experiences in developing the current generation of IO APIs as well as the lessons leveraged that made the current generation a success. For example, we note two example lessons that have been standard techniques for a decade or more and will continue to be relevant into the future. First, buffering will continue to be important. While shifting storage device technology may eliminate the need to work in blocks or in streams for best efficiency, interconnects are still most efficient when moving larger blocks of data. Second, adaptable IO stacks separating the IO API from the implementation, such as the use of MPI-IO or other pluggable transports underneath HDF-5 and PnetCDF and the transport layer for ADIOS, will have continued importance. The shifting landscape and specialized hardware on each new platform require flexibility in how the IO operations are implemented with minimal impact on the application source code. The rise of online workflows that use in-compute-area storage between tasks rather than disks can also be supported with a custom transport layer.

Beyond these two fundamental examples, this paper explores six additional lessons learned as part of developing ADIOS. The lessons are divided into three categories: hardware/infrastructure, API and middleware-related issues, and application support. Each lesson contains suggestions as to how it may be considered when designing new IO stacks for exascale platforms. The paper concludes with some broad thoughts about the lessons presented.

Major ADIOS Contributors

The original core ADIOS team consisted of, in alphabetical order: Hasan Abbasi, Scott Klasky, Jay Lofstead, Karsten Schwan, and Matthew Wolf. This team determined what to build to suit our and our customers' needs. Extensive early input from Robert Ross from Argonne National Laboratory and Garth Gibson from Carnegie Mellon University strongly informed the decisions made and eliminated many potential dead-ends yielding a far better system than could have been done without their input.

Other contributors over the years include Sean Ahern, Ilkay Altintas, Micah Beck, John Bent, Luis Chacon, C.S. Chang, Jackie Chen, Hank Childs, Jong Youl Choi, Alok Choudhary, Julian Cummings, Divya Dinkar, Ciprian Docan, Greg Eisenhauer, Stepane Ethier, Ray Grout, Steve Hodson, Chen Jin, Ricky Kendall, Todd Kordenbrock, Seong-Hoe Ku, Tahsin Kurc, Sriram Lakshminarasimhan, Wei-keng Liao, Zhihong Lin, Qing Liu, Jeremy Logan, Xiaosong Ma, Bronson Messer, Anthony Mezzacappa, Ken Moreland, Ron Oldfield, D.K. Panda, Manish Parashar, Valerio Pascucci, Norbert Podhorszki, Milo Polte, Dave Pugmire, Joel Saltz, Nagiza Samatova, Ramanan Sankaran, Arie Shoshani, Yuan Tian, Roselyne Tchoua, Mladen Vouk, Kesheng Wu, Weikuan Yu, Fang Zheng and Fan Zhang. We thank them for their contributions that have helped refine ADIOS.

2. LESSONS LEARNED

By examining what was learned during the development and subsequent use of ADIOS, we hope to inform the work for IO researchers as they move toward supporting exascale platforms. The lessons are categorized into three groups. First, we present hardware and infrastructure related lessons. These focus on the characteristics of the platforms and how they impact IO. Second are the API level and middleware lessons. These focus on how choices of optimizations per-

formed affect IO performance. Third, we present lessons related to application support. These focus on how to leverage the IO stack in order to support more efficient use of the data. We discuss what each lesson means and the motivations for the lesson as experienced from our work and present a summary of recommendations for future IO work.

Some ADIOS terminology needs to be defined in order to make reading the following lessons clearer:

Group
: A collection of variables and attributes that represent a single output action within an application. For example, "restart", "diagnostics", and "analysis" are common group names.

Process Group
: The output for a particular group by a particular process. While this is generally for a single process, nothing inherent in the design requires that this represent a single process. For situations like aggregation trees that combine the data from several processes into a single output block, a single process group would be written to storage representing all the compute processes aggregated to the single IO process.

Transport Method
: A technique for taking data given by the API and performing some action to process the data. For example, "MPI", "adaptive", "POSIX", and "NULL" use standard MPI-IO independent IO, an adaptive IO technique [13], standard POSIX IO, or ignore the request for output respectively.

2.1 Hardware and Infrastructure Lessons

The hardware and infrastructure lessons focus on the importance of considering the underlying system hardware and system-level software for making a high performance IO stack.

2.1.1 The Storage Stack Matters–A Lot

Initial development of ADIOS was heavily influenced by Lustre and the architecture and usage of Jaguar at Oak Ridge. We were aware of the performance of the interconnect, the interference problems due to competing use of the interconnect and the file system, and the issues of small IO operations on a HDD-based file system. Lustre's ability to select the number of parallel storage targets and the size of each stripe was heavily used to get optimal performance on this platform. In particular, the design of the ADIOS BP file format is built around the ability to set the stripe size in Lustre. The parallel writing performance is organized around manipulating the settings to use as much of the file system as possible. The metadata services bottleneck due to the networking load on the metadata server is also managed. The introduction of the asynchronous IO hints help manage the interconnect limitations. All these decisions were made to work more effectively with the underlying hardware environment. A current technology shift, particularly with regard to the storage array, demands revisiting some of the base assumptions ADIOS made and informs the kinds of issues to consider about the hardware when designing a new IO API.

The deployment of Gordon at the San Diego Supercomputing Center ushered in the era of solid state storage arrays for HPC. While flash-based devices have limitations that will ultimately limit widespread deployment for storage arrays, the cost/performance ratio gap between these devices and hard disk drives has shrunk dramatically in even the last two years. Factoring in the performance and energy use, the differences shrink further. Currently, the price for capacity advantage has shrunk to about 4× [14, 15] for the highest performance models [16, 17]. Previously, HDDs were higher performance than SSDs for streaming operations. SSDs can now reach 392 MB/sec write performance, and HDDs are topping out at 192 MB/sec. For latency, solid state devices have a fixed overhead for accessing any block of data in the device in random order. Performance has shifted to a measure of IOPs (IO operations per second). The IOPs are largely limited by the physical characteristics of the media, just like HDDs.

While flash is currently popular, the relatively low performance compared with that of technologies like phase change memory and the limited write endurance point to an end of life for this technology. Companies like Anobit, purchased by Apple in December 2011, have been working hard to improve the reliability of cheaper MLC-based flash devices further driving prices down and reliability up for the short term. Other efforts to address durability by incorporating rejuvenating operations are also promising [18]. Such advancements will make flash capable of bridging the gap until PCM or other technologies can take over. As these other technologies develop, the attributes governing their performance will be similar to those of flash, but with different performance and durability levels.

Even without using SSDs, paying careful attention to the hardware can lead to performance advantages. Zest [19] offers a "burst buffer" to improve the IO performance. SSD-based burst buffers can offer even better performance [20] between the compute area and the storage array for an HPC machine. Thus greatly improving the perceived performance of IO. Recently offered hardware by Panasas [21] incorporates this technique into a commercial storage array to improve IO performance. Others, such as Starboard Storage Systems and Tegile, offer somewhat similar technology.

The other concern for future technology is that given the high performance of newer solid state storage devices, they likely will be connected using faster busses, such as the memory bus or PCI-express. The performance and cost may even reach a point where current DRAM technology is supplanted by persistent storage, not just for data protection, but to avoid the energy cost of the memory refresh operation. Add this to a multilevel centralized storage array and potentially node-local storage outside the normal address space and the picture for data storage has shifted radically. The additional wrinkle is that at least three or four different performance characteristics must be considered. First is the direct byte-addressable model of DRAM and technologies like PCM. Second is the block-based, but fixed latency, for flash devices. The last characteristics concern disk and tape: different latencies and head movement and streaming, linear access.

The real considerations for IO performance will ultimately be the IO bandwidth to the storage array and the degree of parallelism that can be achieved. Any new platforms will have to more carefully allocate resources for how to specify a storage array and the connectivity to the HPC resource. Achieving a balance of storage bandwidth to interconnect connection to the storage array will be key.

While all modern HPC parallel file systems purport to provide a POSIX [22] interface, different parallel file systems each expose a different level of control to the end user and offer a different level of automation in an effort to enable the best performance. Lustre [23] and PVFS [24] are at one extreme where the end user can control most parameters, such as the stripe size, stripe count, stripe width, and storage target selection. Others, such as GPFS [25] and Panasas [26], consider these decisions too important to allow the end user to manipulate directly. Instead, the file system manages these parameters in creative ways to offer the scalable performance characteristics they exhibit. For example, Panasas dynamically allocates stripes in order to improve the writing performance based on the number of clients writing simultaneously.

Unlike previous standard IO APIs that focused on the data model in the abstract, ADIOS chose to create a data model adapted to the data distributions of parallel file systems. Further, ADIOS manages the file system parameters to control how data is placed on the parallel storage array, when possible. With Lustre in widespread use in the DOE supercomputing centers, ADIOS is able to demonstrate the advantage of this approach on many machines. For example, ADIOS maximizes the number of storage targets and adjusts the stripe width to avoid any false sharing caused by a multiple processes writing to different parts of the same stripe, all without any user intervention.

The BP format developed alongside ADIOS is designed to support this sort of operation. The log-based format affords placing the output of each process at an arbitrary offset in the file that corresponds to the beginning of a stripe. When used in conjunction with Lustre, two main advantages are gained. First, the metadata server hit of spanning to a new stripe is avoided, completely avoiding creating unneeded locks. Second, by maximizing the number of storage targets at file creation time, the maximal parallelization is achieved immediately, rather than adaptively as is done by some parallel file systems. There is one exception in ADIOS. When the three index blocks are written at the end of the file, they are written by a single process in a contiguous chunk without regard to splitting each to a different storage target. The older IO API's focus on the data model did not allow taking advantage of these parameters directly. There is some ability to manipulate them, but not to the extent achieved by the more flexible BP format. The advantage to this approach of manipulating the file system parameters directly is well documented in various ADIOS papers [10, 12, 13].

A side-effect of having more control over data placement on the storage array is the ability to organize the data blocks to afford the best average-case reading performance. The older IO APIs, again, did not offer any real ability to control these limiting the performance advantages. ADIOS has been following the earlier work by the visualization community and others [27–31] to control the organization of data in order to enhance reading performance [32].

Research efforts like active storage [33–35] are effective only if one has knowledge about the data at the file system level. This would require some downward interface to inform the lower layers of the IO stack additional information to support these kinds of operations. If a middleware layer had knowledge of the data sizes and distribution across the compute processes, it could manage the movement rather than being pushed data.

This lesson really motivates a bigger discussion about the semantics of storage. All production parallel file systems offer a POSIX interface and enforce the semantics this requires. Many applications do not need all these semantics and hence suffer performance penalties. Efforts like the Lightweight File Systems Project [36] have explored how to assemble a parallel file system making different consistency semantics optional, for example. Offering this level of flexibility at the file system layer, with the resulting ripples upward, can improve IO performance considerably for particular workloads.

Where exactly to divide layers in the IO stack and what interfaces and functionality to expose in both directions are currently unknown. The shifting technology environment has complicated the picture prompting additional work. Exposing knowledge about both the data and the system characteristics, including negotiation of data organization and placement of operators along the data path, has proven to be an effective approach. Multiple active research projects are continuing to explore how to construct these IO stacks in order to most effectively service the needs of end users.

For now, the ADIOS transport method mechanism has proven effective. New techniques can be tested within the ADIOS framework by using production codes already tuned and configured for ADIOS with no source code changes. This flexibility, combined with the option of optimizing transport methods for particular platforms and situations, makes the ADIOS approach a significant improvement over the fixed IO APIs available today. Validating the ADIOS approach, HDF-5 has begun to offer a similar facility to incorporate a new implementation for how the actual processing of data is performed. The rich API and strict semantics offered by HDF-5 make this more challenging, but offer an excellent path toward the kind of flexibility that will likely be required for exascale systems.

Summary and Recommendations.

Older IO APIs have focused on building the standard data model, with performance a second consideration. ADIOS took a largely opposite approach of focusing on gaining the best parallel writing performance, with more calculation work spent locating the data when it is needed. For exascale platforms, with a complex, multilayered storage hierarchy in which each layer requires different optimization techniques and data layouts, neither of these approaches will be optimal. Introduction of tiers such as node-local persistent storage, data staging, burst buffers [20], and storage devices with different performance characteristics will have to be considered. A holistic approach considering how to generate the best use of parallel resources for data distribution at each level of the hierarchy with a data-layout-aware migration scheme will be required in order to achieve the best overall performance. As data migrates among the various tiers, any decision made to optimize the data layout must be preserved and propagated, potentially requiring data reorganization, in order to ensure continued performance.

The techniques that hide the time spent performing IO can still help address IO performance and are an important part of moving forward. Where these techniques just use asynchronous data movement to hide overheads, re-evaluating how they approach the IO itself is highly encouraged. By aggregating data into fewer locations, the data movement

overhead can be reduced to achieve the canonical data ordering in storage. While in some cases, such as small data sets, this approach still makes sense and where a potentially radically different data distribution may be used, considering the sources of latency in the storage system for both writing and reading is critical. Only when the attributes of these new hardware innovations are taken into account in the APIs will optimal performance be achieved. Incorporating a technique like ParColl [37] that creates appropriately sized canonical blocks in a relatively independent way would improve the performance further.

2.1.2 Asynchronous IO is Getting Harder

Early MPP machines had extensive interconnects that afforded efficient use for simultaneous interprocess communication and asynchronous operations. More recently, the number of interconnect links compared with compute nodes has scaled back relatively to cut down on costs. The downside to this shift is that the proposed approach of using asynchronous IO just at the file system layer to hide the cost of moving data to storage for exascale platforms is unworkable. The DataTap [38] project was motivated by the empirical observation that asynchronous IO for the GTC fusion code on Jaguar at Oak Ridge actually increased the wall-clock runtime 30% beyond what an equivalent run using synchronous IO would take. The key problem identified was a global data exchange phase immediately after the IO phase. By moving the data blocks to storage during that data exchange, the contention on the interconnect dramatically slowed the exchange leading to the performance penalty. This includes removing the time spent performing IO synchronously from the wall clock time. The result was surprising and led to considerable additional work for managing data movement asynchronously. The forecast for interconnect bandwidth for future platforms is no better [39].

The older standard IO APIs were exclusively synchronous. ADIOS also is generally synchronous. Since ADIOS was being created at the same time as this discovery about asynchronous IO, hints were incorporated to help schedule the asynchronous data movement. By offering a begin/end pair around computationally intensive (and communication light) blocks and a pacing indicator, ADIOS enabled scheduling the asynchronous data movement around the general communication phases of the application while pacing the data movement to help ensure minimal blocking due to unfinished data transfers. When selecting an asynchronous IO transport method for a group in the XML file, one can annotate this mapping with the pacing information.

The DataTap system and the follow-on work have focused on how to effectively leverage asynchronous IO to data staging areas with the observation of interference effects that IO-related data movement may have with application-related communication tasks. By incorporating the features described above with DataTap, asynchronous IO has become viable in spite of the reduced capacity of the interconnects.

Summary and Recommendations.

With the diminishing capacity of interconnects compared with the connected nodes, techniques like asynchronous IO are problematic. Current efforts by the MPI Forum and Lustre to incorporate asynchronous IO into their systems can have different levels of effectiveness. By incorporating asynchronous IO into MPI, interprocess communication can be scheduled with higher priority than the bulk data movement operations. Lustre's efforts to rely on asynchronous IO for performance, without the integration with the interprocess messaging system, is likely to repeat the observations that motivated DataTap.

For asynchronous IO to be a viable way to address the time spent performing IO, integration with whatever interprocess communication mechanism deployed is critical. Even taking this into account is not sufficient, however. With multiple applications running on the machine at the same time, coordination among all the running processes prioritizing interprocess communication over bulk data movement operations is necessary. Without this additional level of coordination, achieving consistent performance will not be possible without owning either the entire machine for a single application or at least a relatively isolated partition of the machine.

Using asynchronous IO to aggregate operations that occur successively can avoid some of these problems by delaying performing an operation with the expectation that it can be combined with a subsequent operation reducing the total overhead [40].

Machines like the Blue Gene, with multiple networks that separate storage traffic from interprocess communication, can succeed with asynchronous IO with only minimal special considerations. All other platforms require more global thought to figure out how to move data blocks across the shared interconnect without introducing unacceptably large latencies for interprocess communication.

2.2 API and Middleware Lessons

Increasing efforts are being made in IO middleware to work around limitations of the underlying storage system. ADIOS has enabled this sort of experimentation through the use of the transport methods. These lessons describe the opportunities and challenges faced with using middleware to enhance IO.

2.2.1 Balance Independence and Coordination

Two-phase IO coordinates globally to rearrange data. This communication imposes significant overhead as the process count grows. The general ADIOS approach of every process operating indpendently introduces excessive metadata replication and may create too many small data chunks to offer good reading performance. A balance that offers some coordination, but not too much must be sought.

Avoid Global Coordination.

For the nearly 20 years since the development of collective IO, or more properly techniques like two-phase IO [8] and data sieving [41], global coordination has been promoted as offering the best overall end-to-end IO performance possible. For writing, the reduction in the number of small writes by reorganizing the data into contiguous chunks using the machine interconnect greatly reduces the amount of time taken pushing data to disk. For reading, the rearranged data offered generally fewer reads and more predictable read times.

The concept of reducing the number of operations to the storage system by combining adjacent IOs or using data sieving to perform fewer, larger IO operations is sound. However, keeping the amount of data movement low has proven difficult when applying the two phase approach, especially

in the context of access patterns that exhibit a great degree of data interleaving between processes [37, 42, 43]. Many efforts have been undertaken to optimize the write performance for collective IO [44–46]. While they all address particular scenarios to improve this performance, they do not address the underlying communication overhead. Attempts have also been made to split the output in order to reduce the number of participants with each file improving the scalability [47, 48].

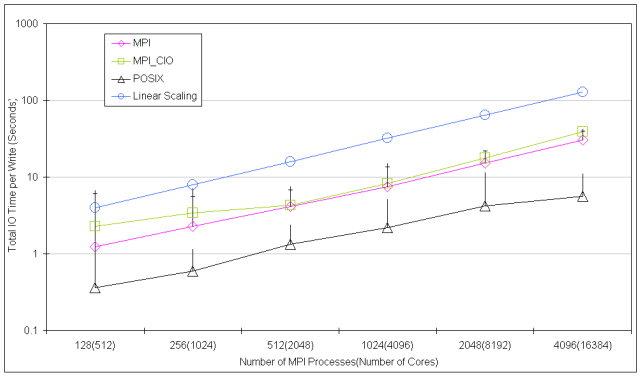

Figure 1: Independent Block Write Performance

ADIOS achieves the goal of fewer, larger IOs with an entirely different approach similar to log-structured storage research [49–53]. Each process writes completely independently from another rather than relying on reorganizing data for a contiguous layout. This approach yields an across-the-board advantage in writing performance [10]. Figure 1 shows the write performance for the GTC [3] fusion simulation with weak scaling for different IO techniques. The POSIX technique illustrated is how ADIOS performs the BP output.

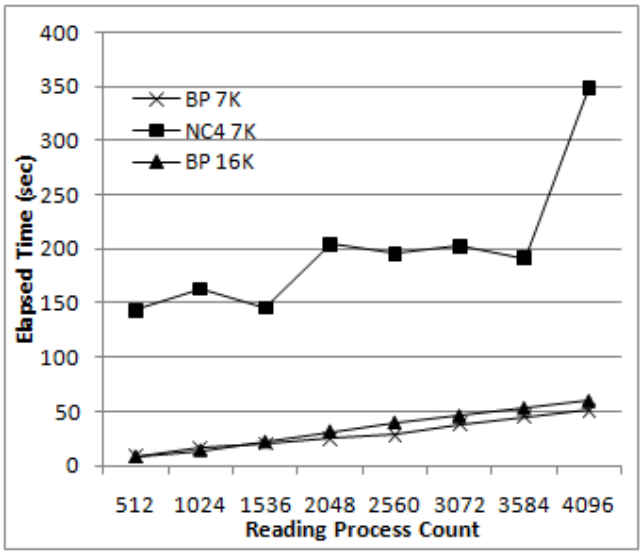

Figure 2: Independent Block Read Performance

This layout generally results in far better reading performance. Figure 2 shows the reading performance for a 3-D domain decomposition written by 7K or 16K processes and read again by the process count indicated. The log format used by the BP format shows dramatically better performance.

While ADIOS does operate as independently as possible, there are two synchronization points during file output generally. First is during the file open when the open calls are partially serialized and the data offsets are collected and distriuted (supported by the adios_group_size call). Second is collecting the local index pieces to the root for writing to the file. All other operations are performed locally only.

The BP format was created to support this level of independence. The log-based format allows each process to own a portion of the file and write all of its local data into that block. While this requires knowing how large the local data will be, in practice this has not been an issue. Adding data to the file is nearly trivial. One just reads the index blocks based on offsets written as the last bytes of the file, writes at any point overwriting the index pieces or not, and then writes new index pieces including the new process groups.

The overall file layout is illustrated in Figure 3. It is a log-based format with the index blocks at the end.

Figure 3: BP Overall File Layout

At a more detailed level the actual layout is far more interesting. Each process group is organized as illustrated in Figure 4.

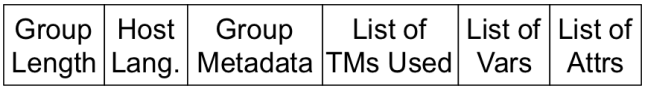

Figure 4: BP Process Group Layout

Each process group consists of the length of this process group, whether or not this process group was written by using Fortran. By storing this flag at this level, ADIOS can support a single file being used by mixed languages. Next is the name of the process group, the coordination var member id (no longer used), the timestamp variable name, and the timestamp value. The final three blocks are replicas of the metadata from the process groups, variables, and attributes, respectively. Each of these blocks is self-contained and only contains replicated data, eliminating any dependence on these blocks for the process group blocks to be valid.

PLFS [54] similarly takes a log-based approach, but the concept is applied in the context of providing a POSIX data mode. Instead of being along the path between the application and storage or requiring a new file format, it is a thin layer on top of the file system that hides how a file is actually written to the file system. Instead of allowing a process to write anywhere, it consolidates all the IO requests from a single process and stores them together. This enables making any IO API independent. On reads, it uses an index to find the requested data. This technique shows excellent performance, but it is too low level to address the whole end-to-end IO performance puzzle. For example, it is too late to intercept the data rearrangement phase that can take so much time at scale [37]. Recently, PLFS has introduced an MPI-IO ADIO layer that hooks in at the collective IO layer, avoiding the data rearrangement phase. This will work well for applications that use MPI-IO. There is also a user-level API that can help address this limitation, but it requires application code change.

Avoid Too Much Independence.

ADIOS demonstrates a nearly extreme level of independence that has helped achieve much of the performance it is capable of achieving. However, a limit to this has been both observed and projected.

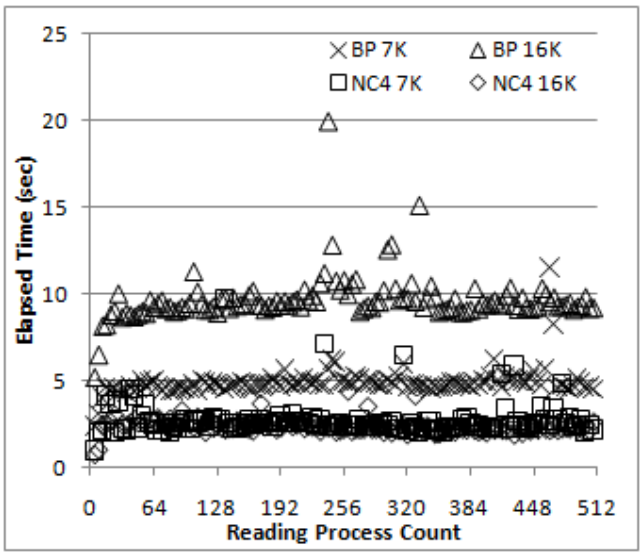

Figure 5: Performance Penalty for Small Process Groups

For example, during reading for small variables, the log-based format can be consistently slower than the comparable standardized data model, as illustrated in Figure 5 and expored in detail elsewhere [12]. The figure shows the performance for reading all of a single variable created from a 2-D domain decomposition with an aggregate size smaller than the memory available to a single process. It is written by 7K or 16K processes using NetCDF4 and ADIOS with a BP format. The NetCDF4 format, by aggregating the tiny pieces into a single whole, is able to avoid numerous small reads and writes, yielding better performance for reading operations.

The key feature that exposes this problem is when a variable is small enough to be fully stored in the memory of a single process. In this case, gathering before writing and scattering after reading is a superior approach. ADIOS is exploring ways to introduce a level of aggregation in order to achieve a middle ground to address this performance gap. The root cause of this gap is the process group structure in the BP file. Since each process group generally represents a single process, the number of locations within a file that must be accessed in order to retrieve a slice of data grows as the computation process count grows. Use of aggregation trees to mitigate the explosion in process groups helps, but is not a full solution. The small data case demonstrates that having special processing for special cases may be required in order to maintain the best performance.

A second problem directly related to the number of process groups is the impact on the index. Currently, the index blocks contain entries for each item in every process group. The log-based structure of the index blocks makes searching these blocks linear with respect to the number of process groups. While the structure currently offers tremendous advantages over searching the actual data sets for data characteristics, the independent entries in the index blocks will eventually fail simply because of the number of entries.

Single Points of Contention Are Bad.

Two examples of the issues with single points of contention can be observed. The first concerns internal causes, such as using a single client process to perform an action on behalf of all other processes. NetCDF's serial processing model was seen as a scalability problem, prompting the move to the NetCDF 4 architecture incorporating collective IO for performance. In spite of the more recent development, ADIOS suffers from a similar single-process bottleneck. The creation of the index blocks at the end of the BP file is done with a single process by collecting and collating all the pieces from all the processes into the three blocks. Unless the index is structured and the process is written to support distributed reading/writing/searching, it is limited to a single process and how much memory it has. ADIOS suffers from this single-process limitation for all the index processing. A related problem is that the index blocks are more of a linear list, prompting the inclusion of an index to the index. For example, an R-Tree [55] would be useful for finding which process groups contain the data of interest during a read operation. Currently, ADIOS has to scan the entire index to find all the entries that contain relevant data. Other research groups are investigating alternative approaches to address this problem.

The second place is external to the application and IO stack. For the current version of Lustre, the limitation of a single metadata server exhibits this sort of single point of contention. Lustre ends up serializing all metadata server requests in order to ensure consistency. One can help manage this load by deliberately slowing the client request rate, although such an approach is counterintuitive. ADIOS demonstrated [10] that with even partial client-side serialization of the open calls can reduce the total time for the open to happen. As explanation, consider the result of all the processes attempting to contact the metadata server simultaneously. It is similar to a distributed-denial-of-service attack. By reducing the rate, the server can process the requests with fewer interruptions to handle incoming connections for additional requests. These sorts of external factors are harder to identify, but can lead to undiagnosed performance problems.

Summary and Recommendations.

ADIOS demonstrated that independence scales far better than using global coordination. PLFS has demonstrated that many of the advantages of independence can be achieved at a low level, but with limits. Any future IO stack should strongly consider how to offer a maximum level of independence. However, it can be taken too far, as illustrated below.

The number of independent pieces (process groups) generated can be a penalty for reading, as well as causing an explosion in space for the index blocks and a corresponding linear increase in time to find items because they have no higher-level structure. The log-based structure of the index blocks is not perfect for all cases. Even when the metadata is very small per entry, the number of entries will eventually hamper any searching. Different structures that offer a balance between the flexibility of log-based formats and coordinated, more structured formats should be investigated.

Additionally, consider the "corner cases" that ADIOS has demonstrably much worse performance and consider the metadata explosion. HDF-5 particularly is known to have special code for small data processing that uses a single process and messaging rather than parallel reading operations. These

sorts of optimizations involve little additional code but can enhance the robustness of any offering. As long as they are dynamic based on current system characteristics, consistent high performance can be achieved for reading, writing, and metadata operations.

Single points of contention both internal to the application and external in other system components, such as the parallel file system, can lead to performance bottlenecks. IO stack developers should consider both the client side and the system components in order to avoid these bottlenecks. As we move toward exascale, even the seemingly most benign aggregation to a single process must be removed or avoided.

2.2.2 *Externalizing Complexity Can Be Good (or Surface Simplicity = Flexibility)*

One complaint users have against the older IO APIs is that they are too complicated and take too much code to do something simple. While from a purely software engineering standpoint these APIs are optimal because they support compile time over runtime checks, end users do not always agree it is the best approach. ADIOS explores the other end of the spectrum by minimizing the surface complexity in an effort to afford greater flexibility. HDF-5 has been moving in this direction as well. Newer releases have offered simpler calls to achieve the same functionality. With a richly complex API, the semantics of what each call actually does is more fixed.

One motivating example prompting the move to simplicity can be seen in the older HDF-5 API versions. For example, explicit opening and closing of the different group levels to write data for a variable were required. If the semantics of data were to change, requiring moving data from one grouping to another, source code changes are required. In the worst case, insertion or deletion of calls might be needed in order to change the navigation through the hierarchy.

The introduction of the companion XML file with ADIOS tested violating strong software engineering for the sake of user convenience and flexibility. The XML file component of ADIOS pulled out all the API calls necessary to configure the metadata about the file into a separate location. The resulting API is only slightly more complex than POSIX. In spite of the separation causing potential undetected incompatibilities and runtime errors, this approach has proved popular with most ADIOS users. As a side effect, additional advantages and challenges were revealed.

The semantics of the POSIX IO API consisting of an open/write/close sequence represents the same model used throughout Linux, with features like ttys and pipes. ADIOS' use of the XML file seeks to achieve the same level of flexibility afforded by the POSIX IO model throughout Linux. ADIOS uses the flexibility to change from writing to disk synchronously to writing asynchronously to writing to a staging area to inserting into an onine workflow. All these possibilities are there because the API is as simple as possible, without any semantics expressed in the code indicating how the IO should be performed. Because all the metadata for the variables is stored in the XML file, moving a variable from one place in the logical hierarchy of the output is also trivial. If the variable name is unique, no changes are required other than to the XML. If not, then if the scripts to generate the API calls are used, then regenerating the calls and a recompile are all that is necessary. Otherwise, changing one line of code in the application should be all that is required.

Although placing the metadata about variables into the XML file has introduced flexibility, it does impose some restrictions. As described above, a group in ADIOS represents an output operation. Since it is defined external to the source code, it needs to be a complete superset of the possible members of the output for every time the output is performed. In general, this is not a problem. A useful feature of this is the ability to tell at a quick glance what are the members of a particular output without isolating and parsing the source code. The inclusion of the utilities in ADIOS that generate the API calls to write, the code necessary for writing can be completely exported to the XML file. However, for applications like S3D [56] that have 50 or more possible variables with only a subset being evaluated for any particular run, the group definition can be needlessly long and have entries for unused variables limiting use of the code generation scripts.

The XML file also enables adding fixed-value attributes to the output without modifying the source code. Some application scientists who carefully evaluate the application with each change prior to any production runs have decided that using an additional path in an added library is unlikely to affect the calculations. This affords adding these attributes once the code has been validated, generating richly annotated output and avoiding revalidation when changing only these annotations.

The most important flexibility introduced by the XML file is the selection of how to manage the data for an output operation. Three different modes are allowed. First, one can change from a to-storage mechanism to a to-staging approach with only a single entry change in the XML file. Second, one can completely turn off a set of output by using the NULL transport method. Third, one can transparently multiplex the output via multiple transport methods. This affords sending data to a persistence mechanism while also forwarding it into an online workflow system. Should the workflow system fail, the persistent copy can be used instead to avoid data loss.

Ultimately, ADIOS introduced an in-code API capable of replacing the XML file. While this API extension is little more than exposing the routines used when parsing the XML file, it did cause some loss of functionality enjoyed by using the XML file approach, but with the benefit of the compile-time checks lost by the introduction of the XML file. Moreover, it removed the requirement of providing a secondary configuration file for the application to operate properly.

Summary and Recommendations.

ADIOS' popularity with early users was driven by their frustration with the complexity of other options. Each user had a story about how much code was required to accomplish seemingly simple IO tasks. The surface simplicity offered by ADIOS ended up affording flexibility that was not possible or extremely difficult with existing approaches.

While older IO APIs have used environment variables and similar mechanisms to afford external control to binary executables, ADIOS explored that space further by exporting all the metadata components into an XML file. This approach has been popular with most users, particularly when the advantages gained are understood. One acn use tools to manage much of the loss of compile time checking, mak-

ing the approach worth serious consideration for future IO stacks.

Making interfaces simple and separating concerns into different components, such as an API and the XML file, offer opportunities not just to simplify the user experience, but also to introduce new options, such as changing how the IO is performed or adding attributes. Future efforts should consider how to maintain simplicity with a nod to modularity so that similar advantages can be achieved.

2.3 Application Support Lessons

With the increasingly large data volumes generated by simulations, reducing the amount of effort searching through raw data is important for reducing the time to insignt. By leveraging the parallel distribution of data prior to writing, embarassingly parallel annotations can greatly reduce the time spent searching the data for relevant portions.

2.3.1 Rich Metadata Enhances Productivity

With data volumes increasing and the cost of moving and scanning data becoming untennable, alternatives should be investigated to aid in data selection. Two categories of metadata should be considered as key features of any new IO stack.

First, for example, one common task performed during analysis for simulation data is determining what the minimum and maximum values are for a variable across the entire domain for each iteration for a data set that has been written to disk. This information can be useful for determining when an energy level or temperature reaches a particular threshold making a particular set of data more "interesting". For older IO APIs, determining the min and max values for a variable requires a full data scan or a noncore extension, such as FastBit [57]. ADIOS incorporates a concept, called data characteristics [10], to support rapid determination of various properties of these massive data sets.

ADIOS' data characteristics are based on the notion that embarrassingly parallel calculations can be used to radically reduce the time spent in determining a data property across a data set. This concept is hardly surprising or new. The only real innovation was incorporating it as a fundamental feature of the BP file format and having it automatically generated as part of the normal ADIOS functioning. The initial implementation of data characteristics focused on simple min/max values. Other unpublished work has evaluated the arithmetic mean and other values. The only real limitation is mathematical: Is it possible to partially calculate a value locally and then use that partial calculation to generate a global value across the entire data set? Applied mathematians have been addressing this question as part of parallelization of algorithms for decades.

The ADIOS index blocks are another level of rich metadata that aids productivity. They incorporate not just the standard name, type, and location metadata, but also the data characteristics for each part of each variable in a process group. These replicate the data from each process group entry to aid in reading performance by affording fewer, larger reads and obtaining larger amounts of data to process. They also afford finishing the partial calculations that the data characteristics represent.

The older standard data files are fully generic and have support for data items that represent both variables and attributes. It is possible to build the same sort of extended features ADIOS has as "standard" by using these formats, but a standard approach has not been adopted by all users of the API, thereby limiting interoperability.

Summary and Recommendations.

Data sizes and the performance bottlenecks of storage interfaces demand that higher-level indexing of scientific data be incorporated into any new IO stack. Further, incorporating a system similar to the data characteristics developed as part of ADIOS affords partial calculations of data properties to accelerate not just finding blocks of data within the storage hierarchy, but also identifying what data is "interesting" and should be read for further analysis. The end-user productivity enhancement and the reduced impact on the file system is tremendous. For example, the data characteristics in ADIOS enables determining the min and max values for a file containing tens of terabytes in nearly the same constant time as a file of a few gigabytes. The time scales as a function of the number of process groups rather than the amount of data. Further, one must consider the costs of global and independent operation and storage as outlined above when developing these features.

2.3.2 Lightly Analyze Data in the IO Path

As was the case with asynchronous IO, managing data movement across the storage interface is important. Asynchronous IO attempts to hide the latency by overlapping the time with other operations. In this case, we seek to avoid the latency entirely by performing some of the lighter-weight data analysis tasks before the data ever crosses the storage interface.

Data staging, or more generally using a small amount of additional resources to accelerate IO in some fashion, has been in use for many years. This was first posited in 1996 to help with imaging for a seismic modeling application [58]. In that work, by adding 10% more nodes, a 30% performance improvement was achieved. It hosted FFT operations to process the data and used asynchronous IO to overlap the IO with computation.

More recently, the DataTap [59] project also incorporated data staging functionality as a way to process the data being moved. Other efforts by Cray [60] and IBM for Blue Gene [61] have also explored the same space. The Nessie [62] project has been used for this purpose as well [43].

More richly, the PreDatA [63] project has demonstrated that hosting functionality along this IO path has different performance characteristics depending on where the operation is placed and the kind of operation. It also shows that using these sorts of operations in the IO path can still improve the total wall-clock time while massaging data into a more desirable form when it reaches disk, even when accounting for the additional resources used for the data staging services.

The DataSpaces [64] project has focused on using data staging as a way to perform code coupling operations. Specifically, it uses asynchronous IO to move data into a staging area and then have a different application retrieve some portion of that data as needed.

A recent effort called Glean [65] at Argonne is a start toward both accelerating IO performance and integrating functionality, such as analysis routines, at the right place transparently. It is similar to PreDatA but extends the location of operations to potentially beyond the current ma-

chine.

Summary and Recommendations.

For most of the cited examples, the staging system hosting the analysis tasks is separate from both the IO API and the storage system. PreDatA and DataTap they take advantage of the transport method mechanism of ADIOS to change how IO is performed. Instead of having data go to storage, a custom transport method moves the data into the staging area for further processing. HDF is incorporating similar functionality by abstracting the internal implementation of how data is written. Glean is incorporating this functionality as a key component in a high-performance IO stack.

Any new efforts for IO stacks, as demonstrated by these examples, should incorporate this functionality. Ideally, new platforms will offer dedicated nodes that offer more RAM or other features that aid these kinds of operations.

3. CONCLUSIONS AND FUTURE WORK

Exascale platforms are projected to have more restricted IO performance compared with that of today's machines. This continues a trend seen with the development of petascale systems from earlier terascale systems. The IO APIs developed prior to ADIOS aimed for a standard data model first and chose to rely on other layers in the IO stack to manage some of the complexity in achieving optimal IO performance. ADIOS was developed acknowledging the achievements and limitations of these earlier IO APIs and achieves higher performance for the petascale environment. ADIOS benefited from the lessons learned from these efforts. The end result is a system that exhibits higher performance for many common IO workloads. By reflecting on the outcomes of the ADIOS project along with those earlier lessons, we can better understand how to architect future IO stacks.

Ultimately, knowledge of all aspects of the storage hierarchy, management of data organization based on size, contents, and the storage mechanism, and annotation of data to aid in selection will make an exascale IO stack effective. Ensuring that limited coordination is performed in order to avoid bottlenecks, but not at the ultimate expense of performance, is also critical.

We hope that the lessons presented here that led to ADIOS, as well as those learned as part of developing ADIOS, will inform future exascale IO stack development.

4. ACKNOWLEDGMENTS

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000. This work as also supported by the U.S. Department of Energy under contract DE-AC02-06CH11357.

5. REFERENCES

[1] M. Christen, N. Keen, T. Ligocki, L. Oliker, J. Shalf, B. Van Straalen, and S. Williams, "Automatic thread-level parallelization in the chombo amr library," Ernest Orlando Lawrence Berkeley National Laboratory, Berkeley, CA (US), Tech. Rep., 2011.

[2] O. E. B. Messer, S. W. Bruenn, J. M. Blondin, W. R. Hix, A. Mezzacappa, and C. J. Dirk, "Petascale supernova simulation with CHIMERA," *Journal of Physics Conference Series*, vol. 78, no. 1, pp. 012 049–+, Jul. 2007.

[3] Z. Lin, T. S. Hahm, W. W. Lee, W. M. Tang, and R. B. White, "Turbulent transport reduction by zonal flows: Massively parallel simulations," *Science*, vol. 281, no. 5384, pp. 1835–1837, September 1998.

[4] R. Rew and G. Davis, "Netcdf: an interface for scientific data access," *Computer Graphics and Applications, IEEE*, vol. 10, no. 4, pp. 76–82, 1990.

[5] J. Li, W.-K. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netcdf: A high-performance scientific i/o interface," in *Supercomputing, 2003 ACM/IEEE Conference*, 2003, pp. 39–39.

[6] "HDF5 home page." http://hdf.ncsa.uiuc.edu/HDF5/.

[7] SILO, "https://wci.llnl.gov/codes/visit/3rd_party/silo.book.pdf."

[8] J. M. del Rosario, R. Bordawekar, and A. Choudhary, "Improved parallel I/O via a two-phase run-time access strategy," in *Proceedings of the IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, Newport Beach, CA, 1993, pp. 56–70, also published in Computer Architecture News 21(5), December 1993, pages 31–38.

[9] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O performance challenges at leadership scale," in *Proceedings ofSC2009: High Performance Networking and Computing*, Portland, OR, November 2009.

[10] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan, "Adaptable, metadata rich IO methods for portable high performance IO," in *Proceedings of the International Parallel and Distributed Processing Symposium*, Rome, Italy, 2009.

[11] M. Polte, J. Lofstead, J. Bent, G. Gibson, S. A. Klasky, Q. Liu, M. Parashar, N. Podhorszki, K. Schwan, M. Wingate, and M. Wolf, "...and eat it too: high read performance in write-optimized hpc i/o middleware file formats," in *Proceedings of the 4th Annual Workshop on Petascale Data Storage*, ser. PDSW '09. New York, NY, USA: ACM, 2009, pp. 21–25. [Online]. Available: http://doi.acm.org/10.1145/1713072.1713079

[12] J. Lofstead, M. Polte, G. Gibson, S. A. Klasky, K. Schwan, R. Oldfield, and M. Wolf, "Six degrees of scientific data: Reading patters for extreme scale IO," in *Proceedings of the Twentieth IEEE International Symposium on High Performance Distributed Computing*. San Jose, CA: IEEE Computer Society Press, Jun. 2011.

[13] J. Lofstead, F. Zheng, Q. Liu, S. Klasky, R. Oldfield, T. Kordenbrock, K. Schwan, and M. Wolf, "Managing variability in the IO performance of petascale storage systems," in *Proceedings ofSC2010: High Performance Networking and Computing*, Nov. 2010.

[14] N. P. P. WD1000DHTZ,

"http://www.newegg.com/product/product.aspx?item=n82e16822236243&tpk=wd1000dhtz," September 2012, wD1000DHTZ, 1 TB, 203.10 MB/sec, $289.99 at newegg.

[15] N. P. P. MZ-7PC12B/WW, "http://www.newegg.com/product/product.aspx?item=n82e16820147163," September 2012, samsung SSD 830, 128 GB, 392.10 MB/sec, $99.99 at newegg [256 MB @ 226.99, 512 MB @ 549.99].

[16] T. H. H. C. S. Writes, "http://www.tomshardware.com/charts/hdd-charts-2012/-25-iometer-2006.07.27-streaming-writes,2929.html," September 2012, wD1000DHTZ, 1 TB, 203.10 MB/sec, $289.99 at newegg.

[17] T. H. S. C. S. Writes, "http://www.tomshardware.com/charts/ssd-charts-2011/as-ssd-sequential-write,2783.html," September 2012, samsung SSD 830, 128 GB, 392.10 MB/sec, $99.99 at newegg.

[18] S. Sharwood, "Flash memory made immortal by fiery heat: Macronix's 'thermal annealing' process extends ssd life from 10k to 100m read/write cycles."

[19] P. Nowoczynski, N. Stone, J. Yanovich, and J. Sommerfield, "Zest checkpoint storage system for large supercomputers," in *Petascale Data Storage Workshop, 2008. PDSW '08. 3rd*, nov. 2008, pp. 1 –5.

[20] N. Liu, J. Cope, P. H. Carns, C. D. Carothers, R. B. Ross, G. Grider, A. Crume, and C. Maltzahn, "On the role of burst buffers in leadership-class storage systems," in *MSST*. IEEE, 2012, pp. 1–11.

[21] P. A. . P. Storage, "http://www.panasas.com/activestor-14," September 2012, 20 HDDs + 10 SSDs for 1600 MB/sec.

[22] IEEE, *2004 (ISO/IEC) [IEEE/ANSI Std 1003.1, 2004 Edition] Information Technology — Portable Operating System Interface (POSIX®) — Part 1: System Application: Program Interface (API) [C Language]*. New York, NY USA: IEEE, 2004.

[23] P. J. Braam, "The lustre storage architecture," Cluster File Systems Inc. Architecture, design, and manual for Lustre, Nov. 2002, http://www.lustre.org/docs/lustre.pdf. [Online]. Available: http://www.lustre.org/docs/lustre.pdf

[24] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur, "PVFS: A parallel file system for linux clusters," in *Proceedings of the 4th Annual Linux Showcase and Conference*. Atlanta, GA: USENIX Association, Oct. 2000, pp. 317–327. [Online]. Available: http://www.mcs.anl.gov/~thakur/papers/pvfs.ps

[25] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proceedings of the USENIX FAST '02Conference on File and Storage Technologies*. Monterey, CA: USENIX Association, Jan. 2002, pp. 231–244. [Online]. Available: http://www.usenix.org/publications/library/proceedings/fast02/schmuck.html

[26] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the panasas parallel file system," in *Proceedings of the USENIX FAST'08Conference on File and Storage Technologies*, M. Baker and E. Riedel, Eds. USENIX, Feb. 2008, pp. 17–33.

[27] V. Pascucci and R. J. Frank, "Global static indexing for real-time exploration of very large regular grids," in *Proc. SC01*, Nov. 2001, pp. 45–45.

[28] H. V. Jagadish, "Linear clustering of objects with multiple attributes," *SIGMOD Rec.*, vol. 19, no. 2, pp. 332–342, 1990.

[29] B. Moon, H. Jagadish, C. Faloutsos, and J. Saltz, "Analysis of the clustering properties of the hilbert space-filling curve," *IEEE T. Knowl. Data En.*, vol. 13, no. 1, pp. 124–141, 2001.

[30] Y. Hu, A. Cox, and W. Zwaenepoel, "Improving fine-grained irregular shared-memory benchmarks by data reordering," in *Proc. SC00*, Nov. 2000, pp. 33 – 33.

[31] S. Kuo, M. Winslett, Y. Cho, J. Lee, and Y. Chen, "Efficient input and output for scientific simulations," in *In Proceedings of I/O in Parallel and Distributed Systems (IOPADS)*. ACM Press, 1999, pp. 33–44.

[32] Y. Tian, S. Klasky, H. Abbasi, J. F. Lofstead, R. W. Grout, N. Podhorszki, Q. Liu, Y. Wang, and W. Yu, "Edo: Improving read performance for scientific applications through elastic data organization," in *CLUSTER*. IEEE, 2011, pp. 93–102.

[33] E. J. Felix, K. Fox, K. Regimbal, and J. Nieplocha, "Active storage processing in a parallel file system," in *Proceedings of the LCI Internaltional Conference on Linux Clusters*, Chapel Hill, North Carolina, Apr. 2005. [Online]. Available: http://www.linuxclustersinstitute.org/Linux-HPC-Revolution/Archive/PDF05/18-Felix_E.pdf

[34] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle, "Active disks for large-scale data processing," *IEEE Computer*, vol. 34, no. 6, pp. 68–74, Jun. 2001. [Online]. Available: http://www.computer.org/computer/co2001/r6068abs.htm

[35] R. Wickremesinghe, J. S. Chase, and J. S. Vitter, "Distributed computing with load-managed active storage," in *Proceedings of the Eleventh IEEE International Symposium on High Performance Distributed Computing*. Edinburgh, Scotland: IEEE Computer Society Press, 2002, pp. 24–34.

[36] R. A. Oldfield, A. B. Maccabe, S. Arunagiri, T. Kordenbrock, R. Riesen, L. Ward, and P. Widener, "Lightweight I/O for scientific applications," in *Proceedings of the IEEE International Conference on Cluster Computing*, Barcelona, Spain, Sep. 2006. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/CLUSTR.2006.311853

[37] W. Yu and J. Vetter, "ParColl: Partitioned collective I/O on the cray XT," *Parallel Processing, International Conference on*, vol. 0, pp. 562–569, 2008.

[38] H. Abbasi, J. Lofstead, F. Zheng, S. Klasky, K. Schwan, and M. Wolf, "Extending i/o through high performance data services," in *Cluster Computing*. Luoisiana, LA: IEEE International, September 2009.

[39] J. Shalf, "Exascale computing technology challenges."

[40] K. Gao, W. keng Liao, A. Choudhary, R. Ross, and R. Latham, "Combining I/O operations for multiple array variables in parallel netCDF," in *Proceedings of 2009 Workshop on Interfaces and Architectures for*

Scientific Data Storage, New Orleans, LA, Sep. 2009.

[41] R. Thakur, W. Gropp, and E. Lusk, "Optimizing noncontiguous accesses in MPI-IO," *Parallel Computing*, vol. 28, no. 1, pp. 83–105, Jan. 2002. [Online]. Available: http://www.mcs.anl.gov/~thakur/papers/mpi-io-noncontig.ps

[42] J. Lofstead, R. Oldfiend, T. Kordenbrock, and C. Reiss, "Extending scalability of collective io through nessie and staging," in *The Petascale Data Storage Workshop at Supercomputing*, Seattle, WA, November 2011.

[43] J. Lofstead, R. Oldfield, and T. Kordenbrock, "Unconventional data staging using nssi," in *In Proceedings of IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, Delft, The Netherlands, May 2013.

[44] R. Bordawekar, J. M. del Rosario, and A. Choudhary, "Design and evaluation of primitives for parallel I/O," in *Proceedings ofSupercomputing '93*. Portland, OR: IEEE Computer Society Press, 1993, pp. 452–461. [Online]. Available: ftp://erc.cat.syr.edu/ece/choudhary/PASSION/sc93.ps.Z

[45] J. Carretero, J. No, S.-S. Park, A. Choudhary, and P. Chen, "Compassion: a parallel I/O runtime system including chunking and compression for irregular applications," in *Proceedings of the International Conference on High-Performance Computing and Networking*, Apr. 1998, pp. 668–677.

[46] R. Thakur and A. Choudhary, "An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays," *Scientific Programming*, vol. 5, no. 4, pp. 301–317, Winter 1996. [Online]. Available: http://www.mcs.anl.gov/~thakur/papers/ext2ph.ps

[47] W. Yu, J. S. Vetter, S. Canon, and S. Jiang, "Exploiting Lustre file joining for effective collective IO," in *Proceedings of the SeventhIEEE/ACM International Symposium on Cluster Computing and the Grid*. IEEE Computer Society, May 2007, pp. 267–274.

[48] K. Gao, W. keng Liao, A. Nisar, A. Choudhary, R. Ross, and R. Latham, "Using subfiling to improve programming flexibility and performance of parallel shared-file I/O," in *Proc. ICPP 09*, Vienna, Austria, Sep. 2009.

[49] T. Blackwell, J. Harris, and M. Seltzer, "Heuristic cleaning algorithms in log-structured file systems," in *Proceedings of the 1995 USENIX Technical Conference*, Jan. 1995, pp. 277–288. [Online]. Available: http://das-www.harvard.edu/users/students/Trevor_Blackwell/Usenix95.html

[50] B. M. Broom and R. Cohen, "Acacia: A distributed, parallel file system for the CAP-II," in *Proceedings of the First Fujitsu-ANU CAP Workshop*, Nov. 1990.

[51] S. Carson and S. Setia, "Optimal write batch size in log-structured file systems," in *Proceedings of the USENIX File Systems Workshop*, May 1992, pp. 79–91.

[52] F. Douglis and J. Ousterhout, "Log-structured file systems," in *Proceedings ofIEEE Compcon*, Spring 1989, pp. 124–129, san Francisco, CA.

[53] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," in *Proceedings of the ThirteenthACM Symposium on Operating Systems Principles*. Pacific Grove, CA: ACM Press, 1991, pp. 1–15.

[54] J. Bent, G. A. Gibson, G. Grider, B. McClelland, P. Nowoczynski, J. Nunez, M. Polte, and M. Wingate, "Plfs: a checkpoint filesystem for parallel applications," in *SC*. ACM, 2009.

[55] A. Guttman, "R-trees: a dynamic index structure for spatial searching," *SIGMOD Rec.*, vol. 14, no. 2, pp. 47–57, Jun. 1984. [Online]. Available: http://doi.acm.org/10.1145/971697.602266

[56] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo, "Terascale direct numerical simulations of turbulent combustion using S3D," *Computational Science & Discovery*, vol. 2, no. 1, p. 015001 (31pp), 2009. [Online]. Available: http://stacks.iop.org/1749-4699/2/015001

[57] E. O"Neil, P. O"Neil, and K. Wu, "Bitmap index design choices and their performance implications," in *Database Engineering and Applications Symposium, 2007. IDEAS 2007. 11th International*, 2007, pp. 72–84.

[58] R. A. Oldfield, D. E. Womble, and C. C. Ober, "Efficient parallel I/O in seismic imaging," *The International Journal of High Performance Computing Applications*, vol. 12, no. 3, pp. 333–344, Fall 1998. [Online]. Available: ftp://ftp.cs.dartmouth.edu/pub/raoldfi/salvo/salvoIO.ps.gz

[59] H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng, "Datastager: scalable data staging services for petascale applications," in *HPDC*, D. Kranzlmüller, A. Bode, H.-G. Hegering, H. Casanova, and M. Gerndt, Eds. ACM, 2009, pp. 39–48.

[60] D. Wallace and S. Sugiyama, "Data virtualization service," in *Proceedings of Cray User's Group*. Cray User's Group, 2008.

[61] J. Fu, N. Liu, O. Sahni, K. E. Jansen, M. S. Shephard, and C. D. Carothers, "Scalable parallel i/o alternatives for massively parallel partitioned solver systems," in *IPDPS Workshops*, 2010, pp. 1–8.

[62] N. S. S. Interface, "https://software.sandia.gov/trac/nessie/."

[63] F. Zheng, H. Abbasi, C. Docan, J. Lofstead, S. Klasky, Q. Liu, M. Parashar, N. Podhorszki, K. Schwan, and M. Wolf, "PreDatA - preparatory data analytics on Peta-Scale machines," in *In Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium, April, Atlanta, Georgia*, 2010.

[64] C. Docan, M. Parashar, J. Cummings, and S. Klasky, "Moving the code to the data - dynamic code deployment using activespaces," in *IPDPS*. IEEE, 2011, pp. 758–769.

[65] V. Vishwanath, M. Hereld, and M. Papka, "Toward simulation-time data analysis and i/o acceleration on leadership-class systems," in *Large Data Analysis and Visualization (LDAV), 2011 IEEE Symposium on*, oct. 2011, pp. 9 –14.